02807
Computational Tools for Data Science

DTU Compute
Department of Applied Mathematics and Computer Science

Heavy Hitters Words in Main Categories

Final Project

Christian Hansen(s146498) & Alex Incerti(s172175) & Avi Szychter(s172275)

December 6, 2018

Contents

1 Introduction

Wikipedia has almost 6 million articles in the English language, and around 1.7 million categories. Some of these categories assigned to the articles are very abstract, and does not tell the reader what the core subject of the article is. However Wikipedia has 25 top-level categories (Refered to as macro-categories)[1], such as *Arts* and *History*, being able to assign one of these categories to an article, would give the reader a better idea of the subject explored in the article. Using these macro-categories, one could find the most common words used within a specific field, and finding these words could then later be used to classify articles that has not been assigned any categories at all.

Our work will focus on assigning one of these 25 macro-categories to an article, and then finding the most common words used in all the macro-categories.

These macro-categories are: *Arts*, *Concepts*, *Culture*, *Education*, *Events*, *Geography*, *Health*, *History*, *Humanities*, *Language*, *Law*, *Life*, *Mathematics*, *Nature*, *People*, *Philosophy*, *Politics*, *Reference*, *Religion*, *Science*, *Society*, *Sports*, *Technology*, *Universe*, *World*

[1] https://en.wikipedia.org/wiki/Category:Main_topic_classifications, accessed: 2018-12-06

2 Problem Statement

The problem that we are going to investigate and try to solve in this project is:

- What are the most common words used in articles within each of the main topic categories?

The intuition here is that the content of each article in Wikipedia falls under one of the macro-categories (Refered to as MCs). For example, intuitively an article about a Country (i.e. Denmark) would belong to *Geography*. Once the articles belonging to each MC are collected, their text can be analysed to highlight the most common occuring words and hence identifying the key-terms characterizing that MC. In practice, this labeling of articles with a MC, is not part of the categorization done by Wikipedia. Articles have finely grained categories which may be mapped back to one of these macro-categories, but this process is not trivial, as it will be explained later.

Here comes a related problem that we need to address, in order to solve the previously explained one:

- How can we assign each article to the most relevant of the main topic categories of Wikipedia?

3 Related works

This section will take a look at work that is related to this project.

3.1 Wikipedia category graph

There has not been many related projects, but there are a few projects that relate to mapping an article to one of the main topic classifications. Most notably an article and implementation done back in 2014 [1]. The project set out to map every article with one of the (at that time) 21 macro-categories. The idea of the article is to make a graph of all the categories within Wikipedia, and then find the shortest path from a given Wikipedia article to one of the MCs. They discovered that most Wikipedia articles get assigned to very few MCs, in fact 50% of all articles gets assigned to just 3 MCs, so it is clear that there is no even distribution between the macro-categories.
The implementation for the project was done using Neo4j, and in the Java programming language. The implementation is open-source and under the Apache2 license, meaning modification and re-distrubtion of the code is allowed. [2]

[2] https://github.com/jacopofar/wikipedia-category-graph, Accessed: 2018-11-22

4 Technologies & Tools

This project utilizes different techniques and tools: the first is a graph database, used to create a graph over all the different categories and then finding the shortest path from every category to one of the 25 macro-categories in Wikipedia. The second is is the Count Min Sktech (CMS), used to identify the heavy-hitters of the words used in these macro-categories. The third is MapReduce, used to parellilize the counting of words for each article.

4.1 Graph databases

A graph database is a type of database which has a focus on creating, storing and retrieving graph structured data in an efficient way. Graph databases derive from *graph theory* in which nodes are the entities, and are connected to each others by edges. Edges have a start and end node. Nodes have properties in the form of key/value pairs [3]. In graph databases, nodes are identified by a unique ID. It is then possible to define a connection (edge) between these nodes with what is called a relationship. [4]. These types of databases are suited for analysing the properties of graphs and usually provide a query language which can express graph specific queries using built-in functions.

Neo4j Neo4j is a company that sells and develops products for graph databases. Especially relevant they provide drivers and API to implement and create graph databases in multiple different programming languages, most notably Java, which has been the most refined and popular [5]. Neo4j provides also a Query Language called Cypher. This query language is SQL inspired and allows to express graph interrogations with an intuitive "*an ascii-art syntax*" [6]. Moreover, it provides specific functions which implement well-known graph theory algorithms, such as the shortest path ones.

4.1.1 Why Graph Databases

A description of the characteristics of a graph database has been provided. This solution has been preferred over the other possible solutions to store and query data, among others: text files and relational databases.

[3] https://whatis.techtarget.com/definition/graph-database, Accessed: 2018-12-06

[4] http://graphdb.ontotext.com, Accessed: 2018-11-22

[5] https://neo4j.com/, Accessed: 2018-11-22

[6] https://neo4j.com/developer/cypher-query-language/, Accessed: 2018-12-06

Text files This type of data storage may well fit some use cases such as logs storage. In these scenarios the amount of data can be very large but it is not often that one needs to perform analysis on that or query the data for specific entities, it is of course possible, and logs may be analyzed to trace back the source of a failure, but it is not an operation performed periodically. *Pros:* text files are very lightweight and do not require any infrastructure. *Cons:* not structured, very slow look-up and query time.

Relational Databases This type of storage is very common and highly structured. Tables map to entities and relationships are mapped using foreign keys and in case of a *many-to-many* relationship, a join table. This type of structure requires many join operations and lookups for additional information in other tables, in case a *many-to-many* type of relationship must be queried. *Pros:* highly structured data storage which makes it easy to query and efficient to lookup, in many scenarios. *Cons:* it does not allow to efficiently map a *many-to-many* relationship and consequently quering it becomes slow and cumbersome [2].

Since our scenario in exclusively working with categories connected with other categories, this is a perfect example of *many-to-many* relationship. Graph databases store data in a way that makes it very efficient to query this type of relationship, since it stores the edges for each node directly along with it. No expensive join operations are needed. For this reason, graph databases fit perfectly our use case. Furthermore, as mentioned earlier, it also provides built-in optimized query functions to query the graph, such as the shortest path.

4.2 Count-Min Sketch

To get the heavy-hitters of the main categories of Wikipedia, the idea is to use Map-Reduce to count all the words in an article, and then use count-min sketch (CMS) to keep track of the heavy-hitters. CMS can be extended, such that it checks whether a given item satisfies a certain fraction (I.e. If more than 1/1000th of items is a specific item) then it is considered a heavy-hitter.

The motivation for using a CMS to store the counts, is that it uses significantly less space (even with a dictionary containing the heavy-hitters), compared to having to store every word in the English language together with how many times it has been seen. We made some calculations for this after the implementation, and it can be seen in Section 6.3. The downside of this is that you are sacrificing precision for space consumption, a CMS only provides an estimated guess of how many times an item has been seen, with the guarantee

that the actual count is never greater than the estimate.

4.3 MapReduce

MapReduce allows for processing large datasets in parallel. The idea being that you can split the workload and distribute it to different processors (or machines), do a few computations and then gather the results. This makes for fast and easy computation.

5 Implementation

This section will describe the implementation done for this project. As it can be seen in the Figure 1, the implementation consists of different steps and elements.

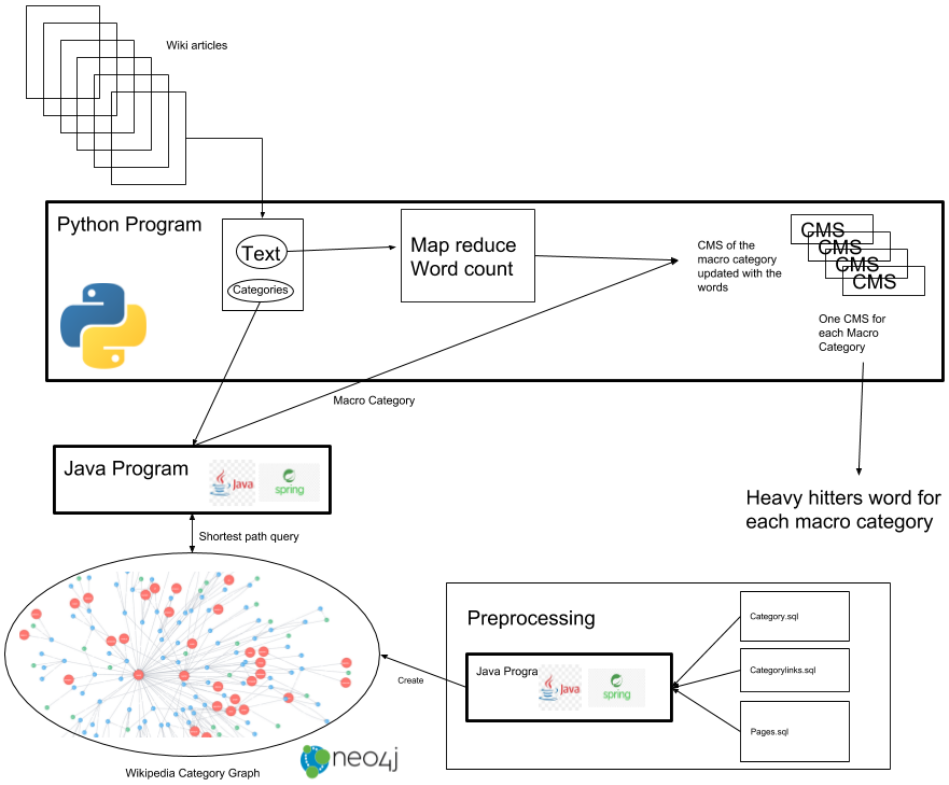

Figure 1: Diagram of the implementation steps and infrastructure (logos taken from [3], [4], [5], [6], [7])

5.1 Graph database

To understand the implementation of the graph database, the reader needs to know the three datasets used. The *category*, *categorylinks* and *page* datasets, linked as shown in Figure 2.

Category table : Contains all categories on Wikipedia.

Categorylinks able : Containss all links to categories. These can be from category-to-category or from article-to-category.

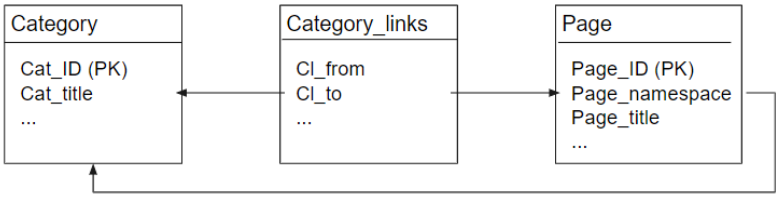

Figure 2: The database schema for the relevant tables

Page table Lists all pages on Wikipedia. Has a field for namespace, namespace 14 is used for categories.

It might seem unnecessary to include all these datasets, as the *page* dataset might seem irrelevant, as the *categorylinks* dataset already has the relationships between categories. But the problem is, that the *categorylinks* defines a link as an id to a title, now we already have all the titles of the pages from the *category* dataset, but the id actually maps to the ids in the *page* dataset, and therefore we need that dataset aswell.

The implementation done in the [1] project, was not exactly what was needed for this project. After investigating and running the code, it seemed to not be working as expected, in fact it seemed not to take into account the *page* dataset and thus creating a graph that was not representing the Wikipedia categories graph that we needed. This could be due to the project being 4 years old, and perhaps Wikipedia has changed in the meantime. Either way, the challenge was now much bigger, as we had to construct the graph database, and create all the queries, connections and labels ourselves.

We chose to keep the language in Java, as described earlier the Java driver is the most common and is seen as the best one. However, the whole project has been re-written using a modern Java framework called *Spring* and its data persistence access module *Spring Data* [7].

We started by creating a node for every category in the *category* dataset. Then we look through the *page* dataset, and map every category with its corresponding id. Then we can created the relationships by going through the *categorylinks* dataset, by getting the actual name from the id in the mappings created with the *page* dataset.

Optimization In order to implement the described procedure in an efficient way, different techniques have been adopted. The writing queries have been

[7] https://docs.spring.io/spring-data/neo4j/docs/current/reference/html/

batched to reduce the expensive database accesses. Batching 10000 writing query statements in only one has improved the running time of more than 10 times. The same goes for the reading of queries. When parsing the *page* file, each entry's name was checked against the graph database, in order to check whether that page's entry was a category and therefore storing the key-value pair of name and id. As explained earlier this pair was needed in order to correctly parse the *categorylinks* file. This file has several millions lines and accessing the graph database for each line was not efficient. The queries have been batched in groups of 10000 and executed at once, retrieving all the nodes needed and processed at once. Batching has been used also when creating relationships parsed from the *categorylinks* file. The results of this optimization can be seen in Figure 4 and have made possible to create the whole graph database in a couple of hours instead of several days.

5.2 Cleaning up of the Graph Database

Once the complete graph was created, it has been analyzed to see if it was behaving as expected. It has been run a shortest path query from a category to all the macro ones and the shortest of these paths has been selected as the macro category assigned to it. This process has been repeated for hundreds of categories. The results showed that most of these categories got mapped to only one macro category. The reason was that there were some *maintenance categories*[8] with a very high degree (number of edges) and this was biasing the shortest path as most of the categories could find a fast way to one of the macro ones through these maintenance ones. It has been performed a cleaning of the graph database, removing these nodes with high degree and no topic related function. Amongst others these categories have been removed: *Tracking categories*, *Noindexed pages*, *Underpopulated categories*, *Container categories*, *Commons category with local link different than on Wikidata*, *Commons category without a link on Wikidata*, *Categories requiring diffusion*, *Hidden categories*

5.3 Mapping one article to the relevant macro-category

The purpose of the graph database is to allow us to map an article to one of the macro categories, as shown in Figure 3. Now that the graph database is

[8] https://en.wikipedia.org/wiki/Category:Wikipedia_maintenance, accessed: 2018-12-06

created, it can be queried and additional information can be gotten, such as creating a query to get the shortest path from two nodes.

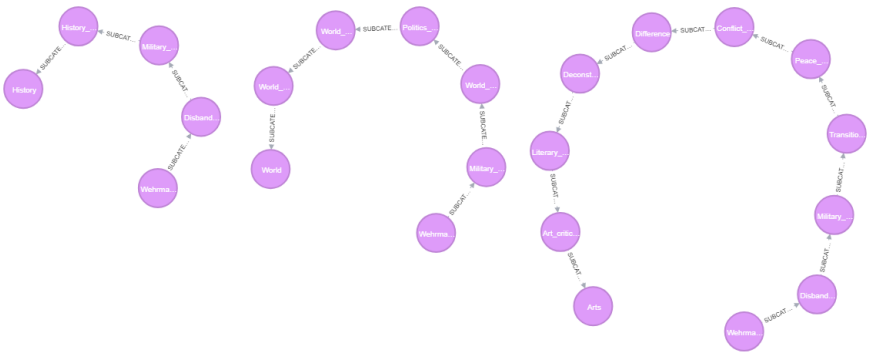

Figure 3: A category's shortest path to Arts, History and World. In this case the mapping would select History as macro categroy assigned

Algorithm & Implementation Each article has multiple categories assigned. For each of these the shortest path to all the 25 main categories has to be computed and the shortest one returned along with the length of the path. Once every article's categories has been mapped to its MC, the MC with most mappings is picked as the MC the article is assigned to. In case of ties the MC with the overall shortest distance, obtained by summing the lengths of the paths of the categories mapping to it, is picked. In case of further tie, the first is selected. This mapping algorithm presents the following challenge:

1. There are over 1.7 million categories in Wikipedia, and calculating the shortest-path from each of those to each of the 25 macro-categories, may take up to minutes.

Optimization This has been carried out on two fronts. The first has been at application level: each shortest path has been moved to be performed by an independent thread, using a ThreadPoolExecutor, which allows to reuse threads and minimizing the overhead of spinning up a new one every time there is some computation to be performed on a thread. This parallel execution has allowed to improve the performance almost with a factor equal to the number of threads.

The second front has been the shortest path query itself. Neo4j has a native query language called Cypher. This query language has native calls for graph related queries such as shortest path. However, simply calling that function for all the 25 macro-categories took up to 2 minutes, due to the fact that the graph has almost 2 million nodes. After some optimization such as running

one query for multiple categories at once, the runtime got down to 20 seconds. The final improvements came from the article [8]: first a loading of the start and end nodes is done using an indexed property, then the shortest path is calculated for all the starting and ending nodes setting a limit to the length of the path. This allowed the query to run in around 4 seconds. In our case an index on the name was already created and the max length of the path has been set to 20 (prior computations showed that the shortest path never got longer than 8). The improvements in performance can be seen in Figure 4.

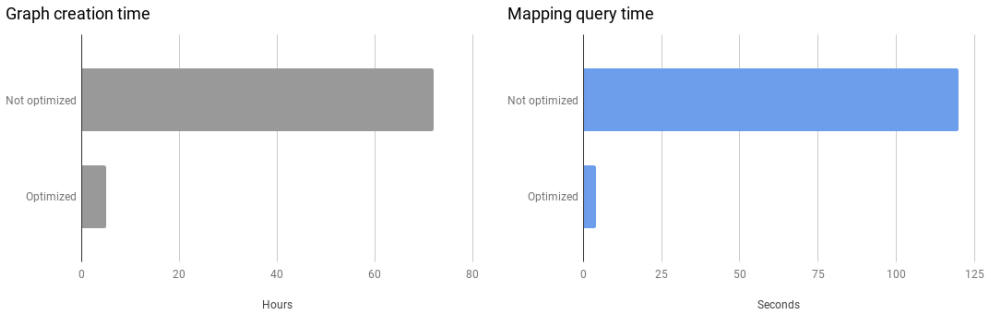

Figure 4: Optimized and non-optimized performance compared. On the left the creation time for the category graph. On the left the time required to map one category to the closest MC.

HTTP interface The Java program exposes an HTTP interface in order to allow external programs to obtain the mapping for articles. The input to the interface is the list of categories an article has and the output is the MC the article has been mapped to, using the algorithm previously explained.

5.4 Heavy-hitters

To find the heavy-hitters, we implemented a python program that consists of multiple parts. This is also shown as an illustration in Figure 1, this program can be seen as the "main" program, not because of complexity, but because it is the program that gathers all the results regarding heavy-hitters. The Python program goes through each article in a dataset one-by-one, for each article it does a word count of the text, queries the HTTP interface and then for every word increments a count-min sketch (CMS) that also keeps track of heavy-hitters. We will now go into more detail with these parts.

Word count

We implemented the word counting MapReduce algorithm. It takes a file as input, this file contains text of the article that we want to count. We go through the file, and get all the words using regular expression. Now the issue is, that if we just use every word in the article text, then the biggest heavy-hitters will most likely become words like "I", "the", "and", "is" and similar words, these are called stopwords and are removed from our word counting. These words have been put into one big list by the python package *nltk*. In addition to these stopwords, we also found that certain words appear a lot in the Wikipedia articles, such as "author", "page", "title", "name", "journal", "publisher", "url" and "http", the reason is that these words are used to describe metadata for links, references and images embedded into the article. They are not described using html-tags or xml, so they do still get picked up by our regular expression. We therefore had to add certain Wikipedia specific stopwords to the word counting. The output of the program is a file with every non-stopword and its count as such:

```
"football" => 5
```

Indicating that the word *football* has been seen 5 times in the article.

HTTP interface

The program queries the HTTP interface and uses it as a blackbox, meaning the python program has no notation of the graphDB and the ava program controlling it. We simply gather all the categories in the article and query the HTTP interface using *grequests*, and then we get a response back. The response simply contains the macro-category that we should assign to the article.

Count-min sketch

Beforehand we have created a CMS for all the 25 macro-categories, and now that we have both the word count of the article, together with the assigned macro-category, we can put the pieces together and find the heavy-hitters. We go through the output of the word counting file, and increment the assigned CMS, however many times the word has been seen (normally you would increment by one in a CMS, but we simply repeat the increment operation). Every CMS has a dictionary for heavy-hitters, containing the word and its count. Whenever a word is increment, we check whether the count of the word satisfies a certain fraction (I.e. 1/2000), then the word is considered a heavy-hitter and added to the dictionary. Then every once in a while (I.e.

for every 5000 items incremented) we run through the dictionary and check whether the word counts still satisfies the fraction, if not they are removed from the dictionary.

Extraction of articles

For extracting the articles from the large .xml dump, we used a open-source tool created by a fellow student. We do not use the entirety of the tool, but just part of it to get an iterator that goes through all articles in a dump [9].

[9] https://github.com/MTelling/WikiXMLDumpToJSON, accessed: 2018-12-06

6 Results

This section will show the different results gathered from this project.

6.1 Distribution

We managed to parse a total of 8600 articles, in around 13 hours. The articles were taken from the large .xml dump containing all articles in Wikipedia. We parsed from the beginning of the file, so the results are slightly biased, as they do not represent the entire dataset, but also because the articles are most likely sorted alphabetically.

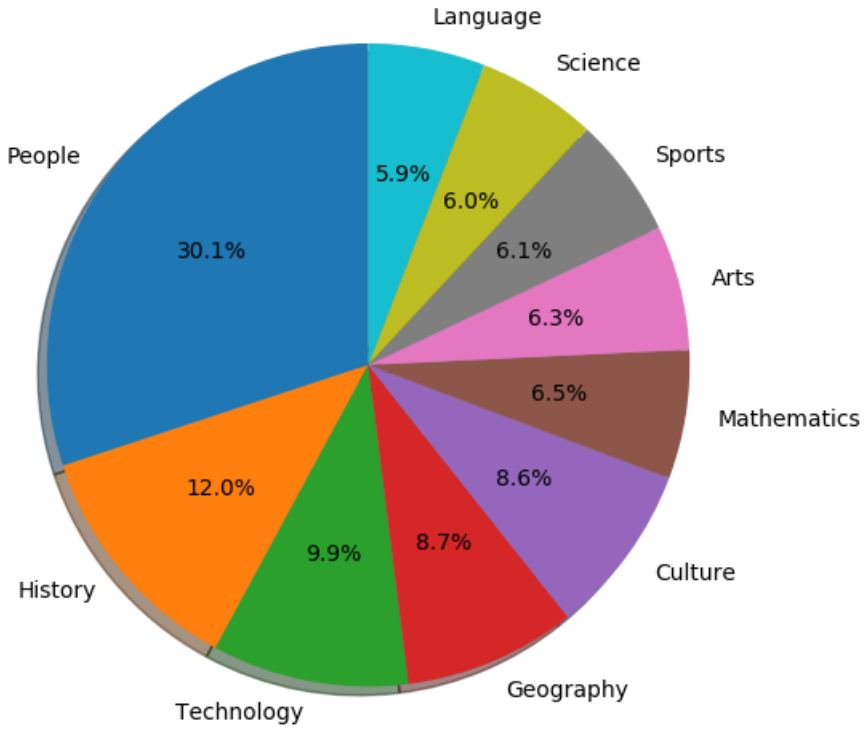

Figure 5: Piechart showing the distribution amongst the 10 largest macro-categories.

Figure 5 shows the distribution of articles mapped to the 10 largest MCs. We decided for this figure, to omit the smallest 15 MCs as the chart gets harder to read and some of them barely have any articles assigned to them. The smallest MCs are *Universe*, *Concepts* and *Reference* which has 12, 14 and 19 articles assigned to them respectively.

Sometimes the assignment is arguable and perhaps not the exact assignment we were hoping for. As an example, the article about the '.NET Framework' got assigned to *Arts* instead of *Technology* or *Science*, the reason being that '.NET Framework' connects to 'Software' which then connects to 'Intellectual Property' which in turn connects to 'Arts', so even though we think '.NET Framework' should be assigned to another MC, when we look at the connections they still make sense. If all articles were parsed the impact of this would be very small, because most articles would be mapped correctly. This is also reflected from our results, which we will go into depth with now.

6.2 Heavy-hitters

We will now take a look at the heavy-hitters for some of the macro-categories. Also note that a CMS estimates the count of a given word, and will often overestimate the actual count. The only guarantee made is that the actual count is greater than the estimated count.

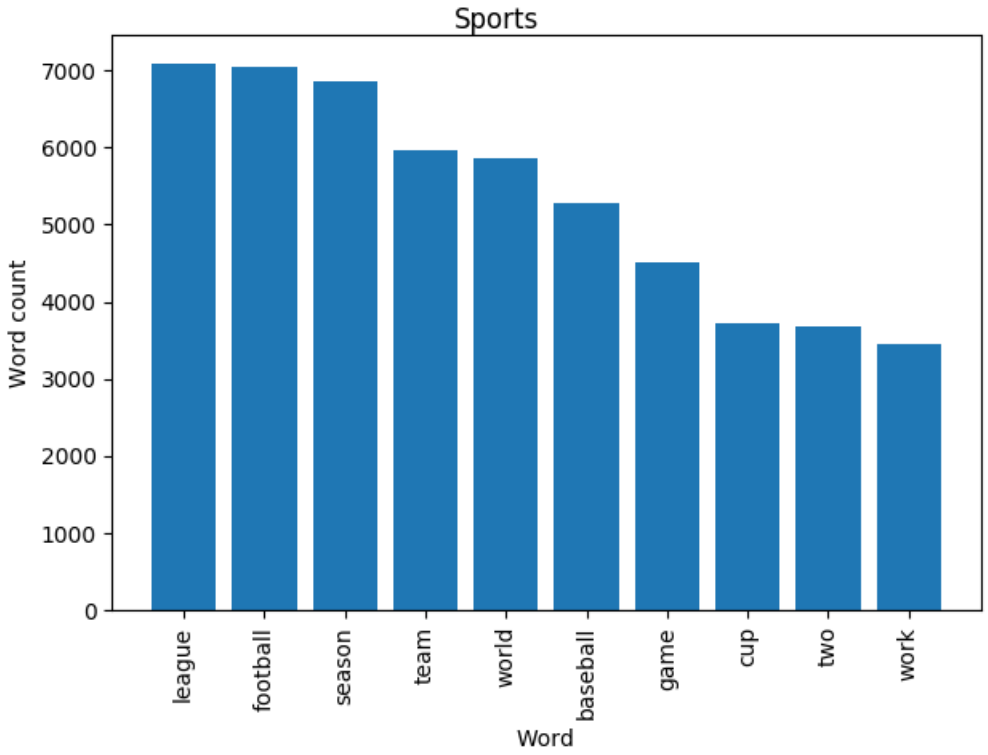

Figure 6: The 10 largest heavy-hitters for the *Sports* macro-category.

As Figure 6 shows, "League", "football", "season", "team", "baseball" and

"game" are heavy hitters for *Technology* and really relevant for the category. "World" seems less relevant when considered alone, but considered together with "cup" (Because of the world cup in football), it starts to make sense, and shows one of the weaknesses of our word counting, we lose context when words are considered alone. The last two words are "two" and "work", which were later found to be words that should probably also be considered stopwords as they appear in pretty much every article, and is a heavy-hitter for most macro-categories. "Work" also shows up as a heavy-hitter for the *Technology* category as shown below.

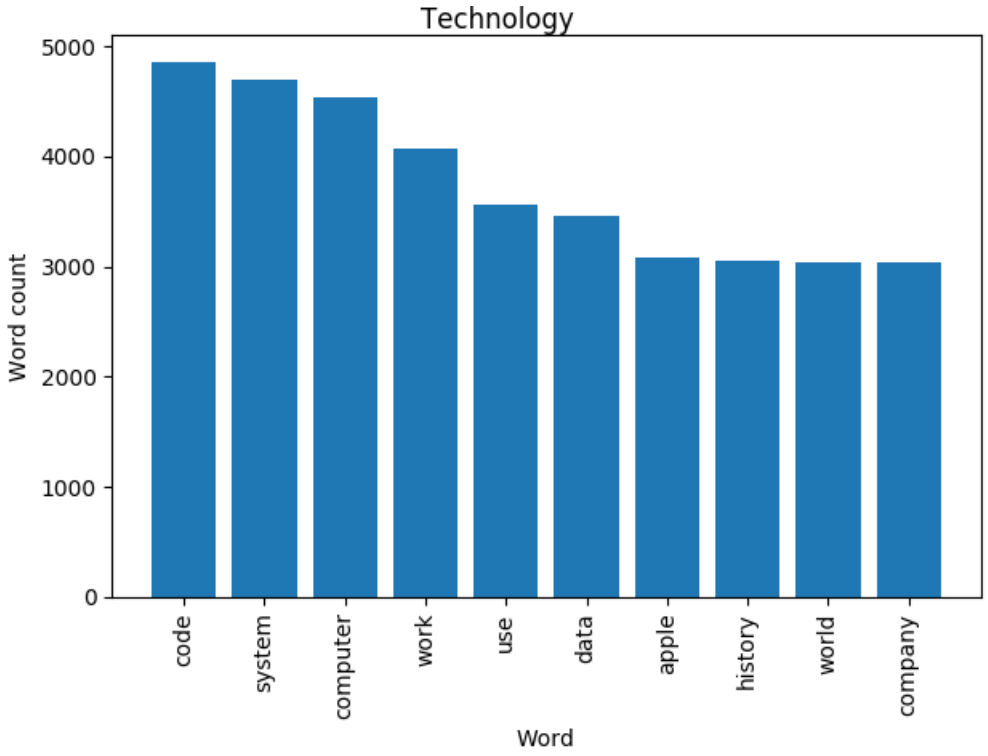

Figure 7: The 10 largest heavy-hitters for the *Technology* macro-category.

Figure 7 also shows the 10 largest heavy-hitters, but this time for the *Technlogy* macro-category. Again we see really relevant heavy-hitters, "code", "system", "computer" and "data", can all easily be related to technlogy. "Apple" might be a bit of a stretch, but it most likely references Apple Inc., due to the fact that we parsed 8600 articles, most likely in alphabetically order, we just happened to parse an article about Apple Inc. We would still expect it to be a heavy-hitter if we parsed all articles, but perhaps not one of the 10 largest.

One interesting thing we can do with these heavy-hitters, is that we can

find the unique heavy-hitters. So the heavy-hitters that only appear in one MC. For technology these **unique heavy-hitters** are "computer", "system", "company", "apple" and "data", interestingly "code" is not unique and after some investigation, we found that it is because it also appears in the *Language* MC. They most likely do not refer to the same "code" (Technology being programming, and in language probably principles or rules), again this shows a limitation that context is not considered or that the same word can have multiple meanings.

6.3 Space Consumption

Finally we analyzed the space consumption of our count-min-sketch solution compared to a more naive but precise one: dictionary. We made a rough estimation of how much space it would have taken to just keep a dictionary with all the seen words and a counter. In this case the counters would have been the exact ones and not an estimastion as in CMS.

Dictionary There are around 170.000 unique words in the english language (our focus is only on the english version of Wikipedia). It is safe to assume that in case we processed all the articles of Wikipedia, we would meet almost all these words and maybe some other ones topic specific and not english, i.e. latin names of plants. We used 170.000 as lower bound estimation. The average length of english words is officially 5, but removing stopwords, which are usually very short and therefore lower the average length, we calculated an average of 6. This estimation has been obtained emprirically by removing the stopwords from some very long articles and counting the average length of the remaining words. In this scenario, the dictionary size for each macro-category would be $170000 * 6 * 4 = 4080000$ bytes (counting 4 bytes for storing an integer as counter). This means around 3.9 Megabytes.

Count Min Sketch In our version of the CMS, we have 27000 columns and 5 hash functions (rows). Each cell stores an integer (the counteer). The heavy hitters list kept by our data structure has on average 265 words contained. Each has an average length of 6 characters, as mentioned earlier, and is stored along with a counter. This gives $27000 * 5 * 4 + 265 * 4 * 6 = 546360$ bytes, which is only 0.52 Megabytes.
Finally, looking at the results of our estimation in Table 1, our solution allows to save more than 80% of the space consumption.

	Dictionary	Count Min Sketch
Space (in Mb)	3.9	0.52

Table 1: Space consumption of the two solution for keeping the heavy hitters for one macro-category

7 Conclusion

The project's aim was to identify the most common words appearing in articles belonging to the 25 main topic classification categories of Wikipedia. This required us to create the graph of categories in order to map each article to the most relevant macro-category. We have analyzed 8600 articles, collecting the corresponding heavy hitters.

Results show that heavy hitters well characterize the macro-category they belong to. Especially the unique-heavy-hitters, which are recurrent only in one specific macro-category, represent an interesting result which may lead to an automatic categorization of unseen articles. Categorizing articles could be done by counting the words in their text and comparing them to the unique-heavy-hitters of the macro-category and assign the article to the MC it shares most HHs with.

Another interesting result which may be derived from the HHs is that their relevance shows that our algorithm for mapping articles, using the shortes tpath and their distance to the MCs, gives relevant mappings in most of the cases. If this was not the case, the HHs we collected would not be as specific and coherent as the ones we were able to obtain.

References

[1] J. Farina, R. Tasso, and D. Laniado, "Assigning wikipedia articles to macro-categories," November 2014.

[2] Neo4j, "Concepts: Relational to Graph." https://neo4j.com/developer/graph-db-vs-rdbms/. [Online; accessed 6-December-2018].

[3] http://www.stickpng.com/img/icons-logos-emojis/tech-companies/python-logo. [Online; accessed 6-December-2018].

[4] https://pngtree.com/freepng/java-programming-icon_3231058.html. [Online; accessed 6-December-2018].

[5] https://s3.amazonaws.com/dev.assets.neo4j.com/wp-content/uploads/20170921035746/note-on-why-native-graph-database-technology.png. [Online; accessed 6-December-2018].

[6] https://www.kisspng.com/png-spring-framework-software-framework-modelview-528 [Online; accessed 6-December-2018].

[7] https://neo4j.com/style-guide/. [Online; accessed 6-December-2018].

[8] M. Hunger, "All shortest paths between a set of nodes." https://neo4j.com/developer/kb/all-shortest-paths-between-set-of-nodes/. [Online; accessed 6-December-2018].